Since $r^2 = \rho^2 + z^2$ it follows that

$$r_\rho = \frac{\rho}{r}, \qquad \theta_\rho = \frac{\cos\theta}{r}, \qquad \phi_\rho = 0.$$

Thus

$$\begin{aligned} u_\rho &= u_r r_\rho + u_\theta \theta_\rho + u_\phi \phi_\rho \\ &= \frac{\rho}{r} u_r + \frac{\cos\theta}{r} u_\theta . \end{aligned}$$

Then we substitute this expression for u_ρ in (A.36) and $\rho = r \sin\theta$ in the denominator of the next to last term of (A.36) to obtain

$$\Delta u = u_{rr} + \frac{2}{r} u_r + \frac{1}{r^2}\left[u_{\theta\theta} + (\cot\theta) u_\theta + \frac{1}{\sin^2\theta} u_{\phi\phi} \right]. \tag{A.37}$$

Spherical coordinates in R^3 centered at a point $\mathbf{x}_0 = (x_0, y_0, z_0)$ with $r = |\mathbf{x} - \mathbf{x}_0|$ are given by

$$x = x_0 + r\sin\theta\cos\phi, \quad y = y_0 + r\sin\theta\cos\phi, \quad z = z_0 + r\cos\theta.$$

The expression for Δu in these coordinates is also given by (A.37).

Appendix B

Elements of MATLAB

In this appendix we give a brief summary of some of the more important MATLAB features that are used in this text. A very convenient short introduction to MATLAB is available in the MATLAB Primer by Kermit Sigmon. A more complete introduction is given in the MATLAB User's Guide, published by Mathworks. To find every possible command, look at the Reference Guide, also published by Mathworks. Even more convenient is the on-line help. For information about any command, function, or function on functions, enter `help` *name*. For example, to find how to use the fast Fourier Transform of MATLAB , enter `help fft`.

In addition there are numerous books published by authors dealing just with the rich subject of MATLAB graphics. Many special application codes using MATLAB have been written and are available at the Mathworks web site: `http://www.Mathworks.com`.

The basic element of the MATLAB environment is the matrix. Scalars are considered to be 1×1 matrices and an n vector is either a row vector (an $1 \times n$ matrix) or a column vector (an $n \times 1$ matrix).

B.1 Forming vectors and matrices

Matrices can be entered by typing in the elements one at a time. Typing this

```
>> [1 2 3;4 5 6]
```

produces

```
ans =

     1     2     3
     4     5     6
```

Notice that we use a semicolon to separate the rows. Usually we want to give a vector or matrix a name. To assign a matrix value to a variable we proceed as follows: Type this

```
>> A = [1 2 3;3 4 5]
```

to get

```
A =

     1     2     3
     4     5     6
```

Many times we do not want to see the value displayed on the screen, especially if the matrix or vector has thousands of elements. This can be accomplished by adding a semicolon after the defining statement.

```
>> A = [1 2 3;4 5 6];
```

If we want to see the numbers in `A`, we enter

```
>> A
```

which produces

```
A =

     1     2     3
     4     5     6
```

The transpose of a matrix is formed by the command `A'`. If the row vector `x` is defined by

```
>> x = [1 5 4 8 10]
```

then `x` is turned into a column vector with the command `x'`. To determine the dimensions of a vector or matrix, use the commands

```
>> size(A)

ans =

     2     3

>> size(x)

ans =
```



```
     1     5

>> size(x')

     5     1
```

To view a certain element in a matrix or vector, we specify its location with a command:

```
>> A(1,2)

ans =

     2

>> x(5)

ans =

    10
```

In many cases the vectors or matrices are far too large to enter one element at a time. For instance if we want to enter a vector **x** consisting of points (0.1, .2, .3, .4, . . . , 5.5, 6), we can use the command

```
>> x = 0:.1:6 ;
```

This row vector has 61 elements. To create a vector of zeros or of ones of the same length there are commands

```
>> y = ones(size(x));
>> z = zeros(size(x));
```

The same works for matrices

```
>> Z = zeros(size(A));
>> Y = ones(size(A))

Y =

     1     1     1
     1     1     1
```

One can also specify a matrix of zeros or ones by giving the dimensions:

```
>> Z = zeros(2,3)
```

The $n \times n$ identity matrix is produced with the command `eye(n)`. There are special commands for entering sparse matrices or diagonal matrices.

B.2 Operations on matrices

Addition and subtraction of matrices are done in the obvious way.

```
>> B = [2 0 -1; 1 2 7];
>> A + B

ans =

     3     2     2
     5     7    13
```

In general, MATLAB can add together only matrices having the same dimension. However, there is one special and very useful exception. If A is a matrix and c is a scalar, then the sum `A +c` means to add c to every element of A. In particular if x is a vector and t a scalar, then `x+t` is a vector of the same length with t added to each component.

Multiplication of matrices is done with the symbol $*$ and presumes that the matrices have the proper dimensions.

```
>> C = [3 3; -1 4; 6 2]
>> A*C

ans =

    19    17
    43    44

>> b = [1 5 -2]
>> A*b'

ans =

     5
    17

>> c = [1 3]
>> c*A

ans =

     3     6     9
    12    15    18
```


A vector or matrix can always be multiplied or divided by a scalar.

```
>> 2 * A

ans =

     2     4     6
     8    10    12

>> A/2

ans =

    0.5000    1.0000    1.5000
    2.0000    2.5000    3.0000
```

B.3 Array operations

Another kind of matrix operation, called array multiplication, is also possible in MATLAB. If A and B are two matrices of the same size with elements $A(i, j)$ and $B(i, j)$, then the command (using `.*` instead of $*$)

```
>> C = A.*B
```

produces another matrix C of the same size with elements $C(i, j) = A(i, j)B(i, j)$. For example, using the same 2×3 matrices A and B we defined earlier, we have

```
>> C = A.*B

C =

     2     0    -3
     4    10    42
```

To raise a scalar to a power, say two, we use the command `5^2`. If we want the operation to be applied to each element of a matrix, we use `.^2`. For example, if we want to produce a new matrix whose elements are the square of the elements of the matrix A, we enter

```
>> A.^2

ans =
```

```
     1     4     9
    16    25    36
```

There is also a kind of array division for two matrices of the same size which divides the two matrices element by element.

```
>> D = [1 3 5; -2 4 -1]
>> A./D

ans =

    1.0000    0.6667    0.6000
   -2.0000    1.2500   -6.0000
```

B.4 Solution of linear systems

Given an $n \times n$ matrix A and an n column vector b, the linear system $Ax = b$ can be solved in several ways. The simplest way is to use the following method.

```
>> A = [1 2 3; 4 5 6; 6 7 9];
>> b = [ 1 0 1]'
>> x = A\b

x =

   -0.0000
   -2.0000
    1.6667
```

The solution to the equation $xA = b$, where x and b are now row vectors, is given by the division command `x = A/b`. Another way to solve $Ax = b$ is to find the inverse of A and then multiply by b.

```
>> C = inv(A);
>> x = C*b;
```

In the first method MATLAB is using the Gaussian elimination procedure with partial pivoting. In general is it more reliable for numerical work requiring high accuracy.

B.5 MATLAB functions and mfiles

MATLAB has the usual built-in functions, such as $\sin x$, $\cos x$, $\tan x$, $\exp x$, $\log x$, $\sqrt{x}$, etc. These functions can take matrices as arguments, in which case the function is applied to each element of the matrix.

```
>> T = [2 3 pi; 8 pi/2 1];
>> cos(T)

ans =

   -0.4161   -0.9900   -1.0000
   -0.1455    0.0000    0.5403

>> sqrt(A)

ans =

    1.0000    1.4142    1.7321
    2.0000    2.2361    2.4495
```

In addition, Bessel functions of various kinds and all orders are available. For example, the Bessel function $J_\alpha(x)$ is called by entering `besselj(alpha, x)`. Another important function is the error function, called simply by entering `erf(x)`. There are also many specialized functions of matrices, such as `eig(A)` which finds the eigenvalues of a square matrix.

It is possible to build more complicated functions using the extremely flexible MATLAB feature known as an *mfile* . There are two kinds of mfiles, *function* mfiles and *script* mfiles. In this section we discuss the former. Script mfiles will be discussed in the next section.

Suppose we need to compute the values of the function

$$f(x) = x\exp(-\sin(x))/(1+x^2).$$

Rather than type this full expression out every time, we can create a function mfile, called `f.m`, so that to evaluate f at $x = 2$, we need only type the command `f(2)`. Here is what the mfile looks like.

```
function y = f(x)
y = x*exp(-sin(x))/(1+x^2);
```

Written this way, the function can take only scalars for x. However, if we write it using the symbols for the array operations, like this:

```
function y = f(x)
y = x.*exp(-sin(x))./(1+x.^2);
```

the function can now be used on vectors and matrices. Notice that in the denominator we are adding the scalar 1 to the vector `x.^2` to produce another vector, which then divides, in array fashion, the factor `x.*exp(-sin(x))`. When a function is written this way, we will say that it is *array-smart*. Array-smart functions are especially important for graphing.

Functions which are defined piecewise may also be constructed in an array-smart fashion. Consider the example

$$f(x) = \begin{cases} x & x < 0 \\ x^2 & 0 \le x < 2 \\ 4 & x \ge 2 \end{cases} .$$

The building blocks for this kind of function are the *characteristic* functions for intervals of the form $(-\infty, a)$ and (a, ∞). An mfile for this kind of function would be

```
function y = c(x)
y = (x < 3);
```

Check that $c(x) = 1$ for $x < 3$ and $c(x) = 0$ for $x \ge 3$. Now we make an mfile for f which is array-smart as follows:

```
function y = f(x)
y1 = x.*(x < 0);
y2 = x.^2.*( (x < 2) - (x < 0) );
y3 = 4*(1 - (x < 2));
y = y1 + y2 + y3;
```

Of course we can also define functions of several variables, such as the fundamental solution of the heat equation in two space dimensions and time.

```
function z = S(x,y,t)
z = exp(-(x.^2 + y.^2)./(4*t) )./(4*pi*t)
```

The variables used in the mfile to define the function are "dummy" variables. One can use any variable names to call the function. For example, for the function f defined above, we can use the statements

```
s = -2:.1:4;
r = f(s);
```

The first command defines the vector **s** with 61 components, and the second command computes another vector **r** with $r_i = f(s_i)$ for $i = 1, \ldots, 61$.

B.6 Script mfiles and programs

Script mfiles are used to collect a sequence of commands that may be lengthy or tedious to type over and over again. Calling the name of the script mfile tells MATLAB to execute the sequence of commands, which we can call a program. In particular script mfiles can call function mfiles. We give an example in which we implement the Euler method for numerically integrating an ODE initial value problem. Let the problem be

$$y'(x) = f(x, y), \qquad y(0) = a.$$

If we wish to compute an approximate solution up to $x = X$, we divide the interval $[0, X]$ into N subintervals of length $\Delta x = X/N$. We label the points where we want to compute the solution as $x_n = (n-1)\Delta x, n = 1, \ldots, N+1$, and we label the computed values y_n. Then Euler's method for computing the y_n is

$$y_n = y_{n-1} + \Delta x f(x_{n-1}, y_{n-1}).$$

Let us take $f(x, y) = x/(1+y^2)$, $X = 10$, and $N = 100$ so that $\Delta x = .1$. Finally, let us take $a = 1$. First we write a function mfile for $f(x, y)$.

```
function z = f(x,y)
z = x./(1+y.^2)
```

Here is a script file to do this integration, call it `xeuler.m`. We use the name `xeuler.m` for " example Euler" because MATLAB already has a program called `euler.m`.

```
X = 10;
N = 100;
a = 1;
delx = X/N;
x = 0: delx : 10;
y = zeros(1,101);
y(1) = a;
for n = 2:N+1
   y(n) = y(n-1) + delx*f(x(n-1), y(n-1));
end
```

The output of this program is the vector $\mathbf{y} = (y_1, \ldots, y_{N+1})$. We start the indexing with one because MATLAB does not allow an index to start with zero. The last

three lines of the file form a *for loop*. Before the loop begins, we set aside the spaces in memory for the vector consisting of $y_1 = a$ and the 100 computed values $y_n, n = 2, \ldots, 101$.

If we wish to make a smaller step size, or change the interval $[0, X]$, or change the initial data, we must edit the file `xeuler.m` and put in new numbers. This can be tiresome if we want to run this program many times with different data. To make the program more interactive, we make X, N, and a input data to be read at run time by replacing the first three lines with the commands

```
X = input('Enter the value of X   ')
N = input('Enter the value of N   ')
a = input(' Enter the initial value a    ')
```

Now when we run the program, it will pause and wait for the data to be entered. After the program has run, we can do further manipulations on the output, such as plotting a curve through the data points. We will discuss plotting 2-D and 3-D graphs in later sections.

B.7 Vectorizing computations

MATLAB is very fast at making vector and matrix calculations. It is less efficient using for loops and these should be avoided when possible. In the program `xeuler`, this could not be avoided because the value of each y_n depended on the previous values. In the following simple example, a for loop can be reduced to a single vector operation.

We want to calculate the sum of the first 100 integers. If we choose not to use the formula $S = N(N+1)/2$, we can do this with the following sequence of commands:

```
s = 0;
for n = 1:100
   s = s+n;
end
s
```

This short program runs very quickly, but we can accomplish the same results without using a for loop as follows:

```
N = 1:100
s = sum(N)
```

The first command is equivalent to `N = 1:1:100` which creates the vector $N = (1, 2, 3, \ldots, 100)$. The second command is a special MATLAB feature which

sums the components of a vector. When applied to a matrix A it sums the columns of the matrix, with output a row vector. The command `sum(sum(A))` yields the sum of all the elements of the matrix. To stress the vectorial aspect of this computation, suppose that we have a function $g(n)$ given by an array-smart mfile `g.m`, and we want to compute

$$\sum_{n=1}^{n=100} g(n).$$

Again we can do this in two lines:

```
N = 1:100;
s = sum(g(N));
```

For a second more complicated example which involves some interesting manipulations of indices, we consider the implementation of a simple finite difference scheme for the linear first-order PDE (see Chapter 2)

$$u_t(x,t) + cu_x(x,t) = 0, \qquad u(x,0) = f(x).$$

If we replace u_t by a forward difference and u_x by a backward difference, we get the finite difference scheme

$$u_{j,n+1} = (1 - c/\rho)u_{j,n} + (c/\rho)u_{j-1,n},$$

where $\rho = \Delta x/\Delta t$. In addition to prescribing the initial data $f(x)$, we shall also assume that the boundary condition $u(0,t) = f(0)$ for all $t > 0$, and we shall solve for u in the set $\{(x,t), 0 \le x \le 10, t > 0\}$. First we write a script mfile that implements this scheme using one for loop inside another. Call this file `fds.m` for "finite difference scheme". We assume that the function f is given by an array-smart mfile `f.m`.

```
c = .75;
delx = .1;
x = 0: delx: 10;
J = 10/delx; % set the number of spatial steps.
delt = .1
r = c*delt/delx;
nsteps = 40; % set the number of time steps.

u = f(x); % initialize the vector u.

for n = 1:nsteps
    v = u;
```

```
    for j = 2:J+1
        u(j) = (1-r)*v(j) +r*v(j-1);
    end
end
```

Notice that we do not have to recompute the first component of the vector u because it is always the same, namely, equal to $f(0)$. The program does not save the values of u at the intermediate time steps. After the program has run, we can view the initial snapshot with the mfile `f.m` and the snapshot at the end of the run, which is the solution at time $t = nsteps \times \Delta t$.

The inner for loop can be eliminated by exploiting an indexing feature of MATLAB. In the program as written, the index j is a scalar. However, in MATLAB it is possible to have vector indices. For example if `k = [2 4 6 7]` then `u(k) = [u(2), u(4), u(6), u(7)]`. Let us take the index j to be the vector $j = (2, 3, \ldots, J+1)$. Using the fact that we can add or subtract a scalar from a vector (section B.2), we see that the vector index $j - 1 = (1, 2, 3, \ldots, J)$. Now we add the statement `j = [2:J+1]` before the outer loop and replace the inner for loop with one vector statement so that the last six lines become

```
j = [2:J+1]
for n = 1:nsteps;
    v = u;
    u(j) = (1-r)*v(j) + r*v(j-1);
end
```

The program is only slightly more compact, but it runs a bit faster. When the dimensions of the problem are quite large, this difference becomes more significant.

B.8 Function functions

MATLAB uses the term "function functions" for a class of functions that operate on functions rather than on matrices. We mention five of them here. The arguments of these function functions consist of strings which name the function operated on as well as various parameters.

The root finder `fzero`. Given a function $f(x)$ defined by an mfile `f.m`, which has a zero in the neighborhood of the point $x = a$, the command `fzero('f',a)` finds the root with a default tolerance of .001. For example, to find the root of $f(x) = x/2 - \sin x$ which lies near $x = 2$, write a function mfile for f and then enter `fzero('f',2)`. It give the answer 1.8955. The call for `fzero` is slightly different in MATLAB 5.0. For more information enter `help fzero`.

The numerical integrators `quad` and `quad8` . These two quadrature routines estimate the value of the integral

$$\int_a^b f(x)dx.$$

`quad` uses an adaptive Simpson's rule, while `quad8` uses an adaptive eight-panel Newton-Cotes rule. The syntax for both of them is the same. Again the function should be defined by a function mfile. However, avoid the use of single letter names like f or h for the functions. For example if the function has the mfile name `ff.m` the command `quad( 'ff', 2,4)` produces an estimate for the integral over the interval [2, 4].

The ordinary differential solvers `ode23` and `ode45`. These two routines numerically solve the initial value problem

$$y'(x) = f(x, y), \qquad y(x_0) = a.$$

`ode23` uses a 2nd/3rd order Runge-Kutta method while `ode45` uses a 4th/5th order Runge-Kutta-Fehlberg method. For example, consider the initial value problem of section B.6, where $f(x, y) = x/(1+y^2)$, $x_0 = 0$, and $a = 1$. First write a function mfile for f, called `f.m`, as we did in Section B.6. Then use these commands with MATLAB4.2 to integrate from $x = 0$ to $x = 5$:

```
>> x0 = 0;
>> xend = 5;
>> a = 1;
>> [x,y] = ode23('f', x0, xend,a);
```

The output consists of the two vectors $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ and $\mathbf{y} = (a, y_1, y_2, \ldots, y_n)$. The numbers y_i are the computed values at the points x_i. The points x_i run from x_0 to x_n but are not usually equally spaced because the method is adaptive. It takes shorter steps when the solution is changing rapidly and longer ones when the solution is changing more slowly.

Both of these routines can be used on systems of ODE's. Enter `help ode23` or `ode45` for more information. In particular, the command to call `ode45` and `ode23` above are correct for MATLAB4.2. However, in MATLAB5.0, the set of ODE solvers has been expanded to include solvers for stiff systems *and* the calling instructions are slightly different. In MATLAB5.0 the last instruction in the program above would be

```
>> [x,y] = ode23('f', [x0,xend], a);
```

Note the addition of the square bracket for the interval of integration.

B.9 Plotting 2-D graphs

MATLAB has an excellent set of graphic tools. In this section we will only touch on some of the most elementary ones. We begin with 2-D graphs. MATLAB takes two vectors $\mathbf{x} = (x_1, \ldots, x_N)$ and $\mathbf{y} = (y_1, \ldots, y_N)$, locates the points (x_j, y_j), and joins them by straight lines. The command is `plot(x,y)`. The vectors $\mathbf{x} = (1, 2, 3, 4, 5)$ and $\mathbf{y} = (2, 3, 1, 5, -1)$ plotted this way produce the picture below (see Figure B.1).

```
>> x = [1 2 3 4 5];
>> y = [2 3 1 5 -1];
>> plot(x,y)
```

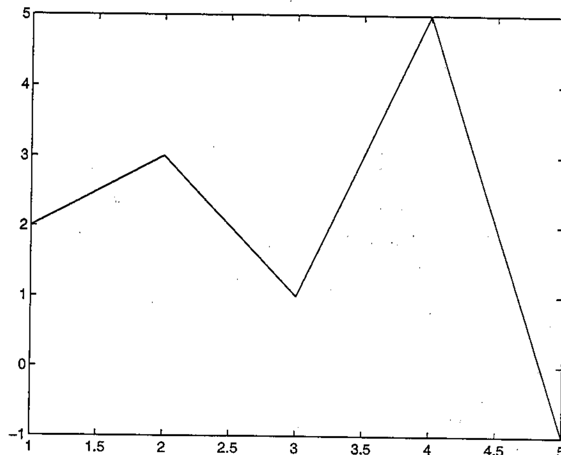

FIGURE B.1
Plot of the vectors $\mathbf{x} = (1, 2, 3, 4, 5)$ *and* $\mathbf{y} = (2, 3, 1, 5, -1)$.

The circle of radius 2, with center at (1, 3) is produced by the commands

```
>> theta = 0:.01:2*pi;
>> x = 1 + 2*cos(theta);
>> y = 3 + 2*sin(theta);
>> plot(x,y)
```

To plot the graph of a function like $\sin x$ on the interval $[0, 2\pi]$, we use the commands

```
>> x = 0:.01:2*pi;
>> y = sin(x);
>> plot(x,y)
```


The last two commands can be combined in one: `plot(x,sin(x))`. If the function f is defined by an array-smart mfile `f.m`, we can also plot it with the command `plot(x,f(x))`.

The output vector **y** from the program `xeuler.m` can be plotted against the vector **x** again with the simple command `plot(x,y)`. This command can be made the last line of the program `xeuler.m` so that the graph is produced automatically.

There are two ways that we can plot several curves on the same graph. Remember that a curve is determined by a pair of vectors **x**, **y** of the same length n. Suppose another pair of vectors **z**, **w** of the same length m where m may differ from n. The first way to plot the two curves on the same graph is with the command

```
>> plot(x,y,z,w)
```

In MATLAB4.2 the first curve will be in yellow, the second in magenta. In MATLAB5.0, the colors will be blue and green.

For example, if we want to compare the initial data $f(x)$ with the snapshot of the solution at time $t = nsteps \times \Delta t$ that results from program `fds` of section B.7, we do it with the command

```
>> plot(x,f(x),x,u)
```

This statement could be appended to the file `fds.m` to make the plot automatically.

Two functions f and g given by array-smart mfiles `f.m` and `g.m` can be plotted on $[-1, 4]$ together with $\exp(x)$ by the commands

```
>> x = -1:.1:4;
>> plot(x,f(x),x,g(x),exp(x))
```

The three curves will be in different colors.

The second way to plot several curves on the same graph uses the command `hold on`.

```
>> plot(x,y)
>> hold on
>> plot(z,w)
>> hold off
```

Both curves will now be same color. The three functions $f(x)$, $g(x)$, and $\exp(x)$ are plotted together these commands.

```
>> plot(x,f(x))
>> hold on
>> plot(x,g(x))
>> plot(x,exp(x))
>> hold off
```

This way of plotting curves together can be combined with a for loop. Put the following statements in a script mfile, call it `graphs.m`.

```
x = 0:.1:20;
plot(x,x.*exp(-x))
hold on
for n = 2:6
    y = x.^n .*exp(-n*x);
    plot(x,y)
end
hold off
```

Now the six curves will be plotted on the same graph by calling the program `graphs`.

Further 2-D graphing features

The `axis` command. When we use the command `plot(x,y)`, MATLAB automatically plots the curve on the rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. If we wish to change this scale, perhaps to expand a portion of the graph, and instead plot on the rectangle $[a, b] \times [c, d]$, we follow the plot command with `axis([a b c d])`. You can return the axis scaling to the automatic, default, mode with the command `axis('auto')`.

The `zoom` command. This is another way to blow up a portion of the graph, using the mouse. Enter the command `zoom on`. Then move the pointer to the region of the graph you want to enlarge. Click with the left mouse button. This will blow up the size by a factor of two. Clicking again blows it up again by a factor of two. Clicking with the right mouse button has the opposite effect. The command `zoom out` restores the original figure. `zoom off` turns off the zoom feature.

The `ginput` command. This feature allows one to pick off the coordinates of points on a figure using the mouse. The command is `[x;y] = ginput(n)`. Move the pointer to n different points in the figure, and click at each point with the left mouse button. When you have clicked on the last point, the coordinates of the n points are displayed on the screen in column vectors **x** and **y**.

B.10 Plotting 3-D graphs

In this section we will see how to graph functions of two variables. The basic plotting variable for 2-D graphs is the vector. In 3-D graphs, the basic plotting variable is the matrix.

Most often we want to plot a function $f(x, y)$ over the rectangle $a \leq x \leq b, c \leq y \leq d$. First construct a mesh over the rectangle by selecting a stepsize in the x direction, Δx, and a stepsize in the y direction, Δy. Then construct the vectors **x** and **y** with the commands `x = a:delx:b` and `y = c:dely:d`. The next command creates two matrices : `[X,Y] = meshgrid(x,y)`. If n is the length of **x** and m is the length of **y**, both X and Y are $m \times n$ matrices. The m rows of X are all equal to the vector **x**, and the n colums of Y are all equal to the vector **y**. A corresponding $m \times n$ matrix of the values of f at the grid points is generated by the command `Z = f(X,Y)`. We assume that $f(x, y)$ is expressed by an array-smart mfile `f.m`. A three-dimensional wire mesh surface is generated by the command `mesh(X,Y,Z)` while a faceted surface is generated by the command `surf(X,Y,Z)`. We could, of course, combine two commands into one as `mesh(X,Y,f(X,Y))` or `surf(X,Y,f(X,Y))`.

For an example, write a function mfile for $f(x, y) = \exp(-(x-3)^3 + (y-2)^2)$.

```
function z = f(x,y)
z = exp(-( (x-3).^2 + (y-2).^2) );
```

To graph f over the rectangle $[0, 4] \times [0, 6]$ use the sequence of commands

```
>> x = 0:.1:4;
>> y = 0:.1:6;
>> [X,Y] = meshgrid(x,y);
>> Z = f(X,Y);
>> mesh(X,Y,Z)
```

Follow this with the command `surf(X,Y,Z)` to see the faceted surface. For another example, try `Z = sin(X-Y)`. The wire mesh surface looks like this (see Figure B.2).

We do not always plot graphs of functions over rectangles. Sometimes the domain of interest for the function is a disk or an ellipse. In this case the matrices X and Y cannot be generated by the simple `meshgrid` command. To generate a grid for the disk of radius a, centered at the origin, we must introduce polar coordinates. Let Δr and $\Delta\theta$ be the spacing of the mesh in the radial and angular variables. Then we construct the X and Y matrices for polar coordinates with these commands.

```
>> r = 0:delr:a;
>> theta = 0:deltheta:2*pi;
>> X = r'*cos(theta);
>> Y = r'*sin(theta);
```

In the last two lines we take the matrix product of the column vector which is **r** transpose and the row vectors $\cos\theta$ and $\sin\theta$. These operations produce full matrices. To see the mesh of coordinates in the x, y plane use `mesh(X,Y,Z)`, where `Z = zeros(size(X))`. Now try `mesh(X,Y,X+Y)` and `mesh(X,Y,sin(pi*(X.^2 +Y.^2)))`. The result is shown in Figure B.3.

If the function f to be graphed is given as a function of polar coordinates $f = f(r, \theta)$, we must construct matrices R and Θ to use as the arguments for f. We want all the rows of R to be the vector **r** and all the columns of Θ to be the vector θ. This is accomplished by taking the matrix products

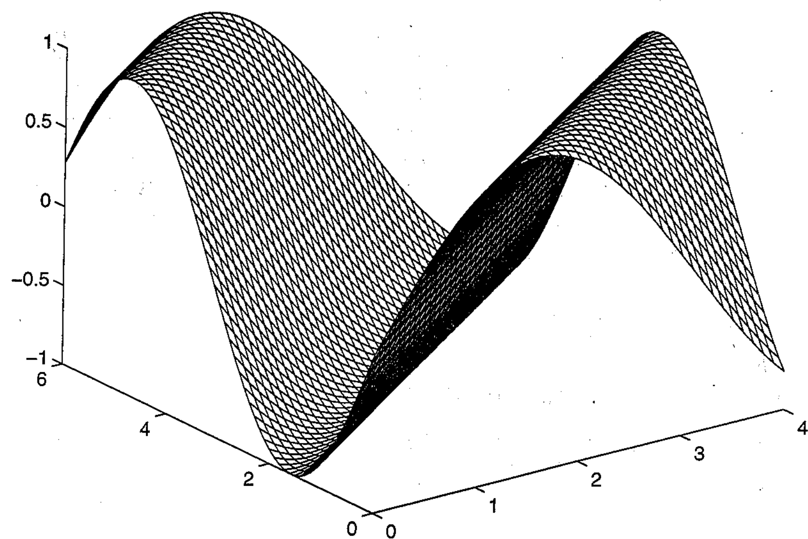

FIGURE B.2
Mesh surface of $z = \sin(x - y)$.

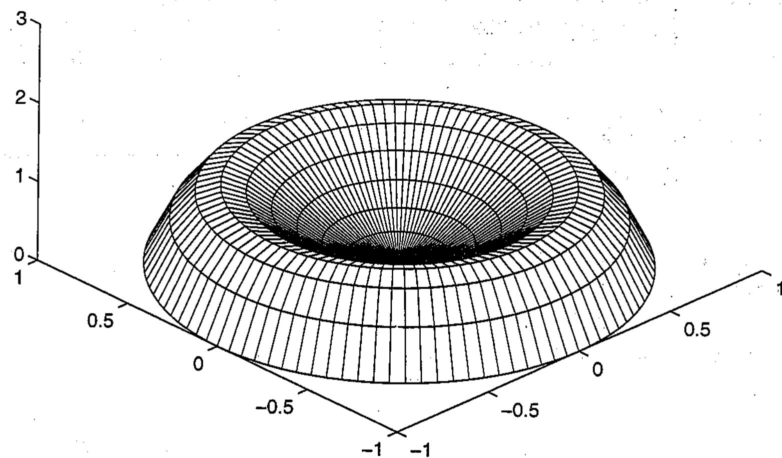

FIGURE B.3
Plot of $f(x, y) = \sin(\pi(x^2 + y^2))$ *over the unit disk.*

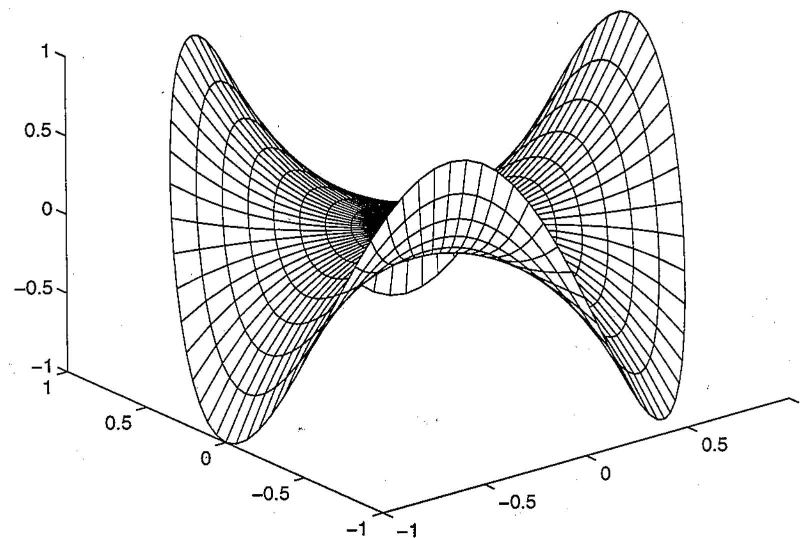

FIGURE B.4
Graph of $f(r, \theta) = r^3 \cos(3\theta)$ *over the unit disk.*

```
>> R = r'*ones(size(theta));
>> TH = ones(size(r))'*theta;
```

Now we can plot the function $f(r, \theta) = r^3 \cos(3\theta)$ over the disk of radius a with the command `mesh(X,Y,Z)` where `Z = R.^3.*cos(3*TH)`. The result is shown in Figure B.4.

Further 3-D graphing features

The `axis` command has a 3-D analogue. To set the scaling with limits a, b for the x axis, c, d for the y axis, and e, f for the z axis, use the command `axis([a b c d e f])`.

The `view` command. The direction from which a 3-D graph is viewed is specified by two angles: the azimuthal angle, which is the angle of rotation from the negative y axis, and the angle of elevation above the x, y plane. For example the view down the positive x axis toward the origin has azimuthal angle of 90° and elevation of 0°. The default direction for the 3-D graphs has angles $(-40, 30)$. To change the view enter, for example, `view(45,50)`. Try the `view` command with the graph of Figure B.2.

The `pcolor` command. Instead of a 3-D surface plot or mesh plot, the command `pcolor(X,Y,Z)` makes a 2-D plot of the colors used in `surf` or `mesh`. This is especially useful in visualizing heat flow. It can also be combined effectively with the next command.

The contour command. To see the level sets (contour lines or curves) of a function $f(x, y)$, proceed as in the surf or mesh command to make meshgrid matrices and a function value matrix. Then contour(X,Y,Z,n,'k') will divide the interval [min f, max f] into $(n + 1)$ equal subintervals and plot n contour lines. The 'k' means they will be plotted in black. Other colors are possible.

B.11 Movies

Movies are constructed by using a for loop and the getframe command. Here is a sample program to see the vibrations of the lowest mode of the vibrating string on the interval [0, 1]. The solution is the function $u(x, t) = \sin(\pi t) \sin(\pi x)$. We shall make a movie which goes through one full period of the motion, which in this case is $T = 2$. Then by running the movie several times, we will be able to see the periodic vibrations. Let the movie have 20 frames, i.e., one snapshot per time interval $\Delta t = .1$. Each snapshot will be stored as a column of a matrix, call it M, as follows:

```
x = 0:.1:10;
moviein(20);
for j = 1:20
    t = (j-1)*.1
    u = sin(pi*t)*sin(pi*x)
    plot(x,u)
    M(:,j) = getframe;
end
```

The getframe command make a scan of the figure that results from the plot command. The length of columns of M depends on the size of the figure. If you enlarge the figure, the matrix M will be larger, requiring more memory.

Now that the movie is made, we can run it with the command

```
>> movie(M,4,10)
```

The 4 is the number of times the movie will be run, in this case 4 complete periods. The 10 is the number of frames per second. If there is no third argument, the movies runs at the default speed of 12 frames per second.

Appendix C

References

[A] S. Antman, The equations for the large vibrations of strings, *American Mathematical Monthly*, **87**(1980), 359-370.

[AS] M. Abramovitz and I. Stegun, *Handbook of Mathematical functions*, Dover Publications, New York, 1964.

[Bar] R. Bartle and D. Sherbert, *Introduction to Real Analysis*, Wiley, New York, 1982.

[BaS] I. Babuska and B. Szabo, *Finite Element Analysis*, Wiley-Interscience, New York, 1991.

[BD] W. Boyce and R. DiPrima, *Elementary Differential Equations*, sixth edition, Wiley, New York, 1997.

[Be] J. Benedetto, *Harmonic Analysis and Applications* CRC Press, Boca Raton, Fl., 1996.

[Bo] F. Bowman, *Introduction to Bessel Functions*, Dover Publications, New York, 1958.

[BH] W. Briggs and V. Henson, *The DFT, An Owners Manual for the Discrete Fourier Transform*, SIAM, Philadelphia, 1995.

[BS] S. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*, Springer-Verlag, New York, 1994.

[CF] R. Courant and K.O. Friedrichs, *Supersonic Flow and Shock Waves*, Springer-Verlag, New York, 1948.

[CourHilb] R. Courant and D. Hilbert, *Methods of Mathematical Physics*, Vols. 1 and 2, Wiley-Interscience, New York, 1962.

[ChowHale] S. Chow and J.K. Hale, *Methods of Bifurcation Theory*, Springer-Verlag, New York, 1982.

[CL], E. Coddington and N. Levinson, *Theory of Ordinary Differential Equations*, McGraw-Hill, New York, 1955.